VI.01. Интерфейсы

Что такое интерфейс?

По крайней мере, что такое интерфейс в Go? Формально говоря, интерфейс - это набор сигнатур методов; сигнатура метода - это, грубо говоря, *заголовок метода, включает в себя имя метода, типы и порядок следования параметров и возвращаемых значений.* Интерфейсы позволяют определять функции, которые не привязаны к конкретной реализации. То есть интерфейсы определяют некоторый функционал, но не реализуют его.

Замечание в сторону (не для детей): немного похоже на ООП с его перекрыванием методов при наследовании, а интерфейс - это что-то вроде абстрактного класса - методы определены, но не реализованы.

Иначе говоря, интерфейс говорит **какие действия может делать** объект, но не что у него внутри, не затрагивает его устройства.

Когда тип предоставляет определения для всех методов в интерфейсе, говорят, что он реализует интерфейс. Интерфейс определяет, какие методы должен иметь тип, а тип решает, как реализовать эти методы.

Есть такая фраза: если нечто мяукает как кошка, лакает молоко как кошка, ловит мышей как кошка, умывается как кошка, то это, скорее всего, кошка. Мы охарактеризуем ситуацию немножко иначе. `Кошка` является интерфейсом с сигнатурами методов `Мяукать()`, `Лакать_молоко()`, `Ловить_мышей()`, `Умываться()`. Любой тип, который предоставляет определения для методов мяукания, лакания молока, ловли мышей и умывания, грубо говоря, является кошкой, говоря более точно реализует интерфейс кошки. И неважно, это кошка Кася, робокот из мультика или набор битов, который представляет переменную такого типа - все они умеют делать все действия интерфейса `Кошка`, и, значит, реализуя интерфейс `Кошка`, являются с нашей точки зрения кошками.

~~Например, `Стиральная_машина` может быть интерфейсом с сигнатурами методов `Стирка()`, `Полоскание()` и `Отжим()`. Любой тип, который предоставляет определения для методов стирки, полоскания и отжима, грубо говоря, является стиральной машиной, говоря более точно, реализует интерфейс стиральной машины.~~

Какое-то общее представление, направленность нашего движения определена. Прежде чем переходить к тому где, как и для чего применять интерфейсы, давайте поговорим,

разумеется, о техническом моменте.

Объявление и реализация интерфейса

Что-то подобное у нас было в первом семестре - мы искали в строке цифры. Вот давайте такой интерфейс и напишем: объявим интерфейс DigitsFinder и реализуем его

пример 01a.go

```
package main

import (
    "fmt"
)

//interface definition
type DigitsFinder interface {
    FindDigits() string
}

type MyString string

//MyString implements DigitsFinder
func (ms MyString) FindDigits() string {
    var digits []rune
    for _, r := range ms {
        if r >= '0' && r <= '9' {
            digits = append(digits, r)
        }
    }
    return string(digits)
}

func main() {
    var d DigitsFinder
    date := MyString("Friday, 30 July 2021. 12:05:35")
    d = date // possible since MyString implements DigitsFinder
    fmt.Printf("Digits are %s\n", d.FindDigits())
}
```

выводит

```
Digits are 302021120535
```

В программе создан интерфейсный тип `DigitsFinder`, имеющий единственный метод `FindDigits() string`.

Затем объявлен тип `MyString`.

И далее для ресивера типа `MyString` добавлен метод `FindDigits() string`. Все методы (хоть он и один единственный) интерфейса `DigitsFinder` реализованы в типе `MyString`, тем самым тип `MyString` реализует интерфейс `DigitsFinder`.

В main() мы заводим переменную `d` интерфейсного типа `DigitsFinder` и присваиваем ей значение переменной `date` типа `MyString`. Это возможно, поскольку тип `MyString` реализует интерфейс `DigitsFinder`. Да, конечно, можно было не объявлять переменную `date`, можно было просто написать так:

```
    d = MyString("Friday, 30 July 2021. 12:05:35")
```

но нам это пригодится в дальнейших рассуждениях.

И в следующей строке мы вызываем метод `d.FindDigits()`, причём вызывается именно метод `FindDigits()` типа `MyString` (ведь переменная `d` содержит величину именно типа `MyString`, ну, и выводим результат.

Ок, мы создали и реализовали интерфейс (интерфейсный тип).

И что с этим делать?

И вот теперь займёмся чуть более содержательными действиями с интерфейсами. Ведь в приведённом выше примере интерфейс был не очень-то и нужен - мы ведь могли просто вызвать метод `DigitsFinder` для переменной `date`: вместо `d.DigitsFinder()` написать просто `date.DigitsFinder()`. Давайте рассмторим пример, в котором интерфейс уже играет определённую роль:

пример 01b.go

```
package main
```

```go
import (
    "fmt"
    "math"
)

//interface Shape definition
type Shape interface {
    Area() float64
    Perimeter() float64
}

//Shape's implementations: Rectangle, Circle
type (
    Rectangle struct {
        Width  float64
        Height float64
    }
    Circle struct {
        Radius float64
    }
)

//Rectangle implements Shape ...
func (r Rectangle) Area() float64 {
    return r.Width * r.Height
}

func (r Rectangle) Perimeter() float64 {
    return 2.0 * (r.Width + r.Height)
}

// ... as well as Circle
func (c Circle) Area() float64 {
    return math.Pi * c.Radius * c.Radius
}

func (c Circle) Perimeter() float64 {
    return 2.0 * math.Pi * c.Radius
}

func main() {
```

```
    var s Shape
    s = Rectangle{Width: 5.0, Height: 7.0}
    fmt.Printf("%T %v\n", s, s)
    fmt.Printf("area = %g, perimeter = %g\n", s.Area(), s.Perimeter())
    s = Circle{Radius: 3.5}
    fmt.Printf("%T %v\n", s, s)
    fmt.Printf("area = %g, perimeter = %g\n", s.Area(), s.Perimeter())
}
```

выводит

```
main.Rectangle {5 7}
area = 35, perimeter = 24
main.Circle {3.5}
area = 38.48451000647496, perimeter = 21.991148575128552
```

Здесь мы объявляем интерфейсный тип (интерфейс) Shape с методами вычисления площади Area() float64 и периметра Perimeter() float64, который и реализуем, причём два раза: через типы Rectangle и Circle. Самое забавное здесь то, что одна переменная s интерфейсного типа Shape принимает значения двух разных типов, реализующих этот интерфейс - это хорошо видно на выводе. Т.е. можно говорить (и так и говорят, хоть это и формально не совсем точно, зато хорошо передаёт смысл, что и Rectangle, и Circle являются Shape'ами).

Теперь давайте рассмотрим **ещё один пример** с чуть-чуть более "практическим" использованием интерфейса.

Напишем простую программу, которая рассчитывает общие расходы для компании на основе индивидуальных зарплат сотрудников. Для простоты предположим, что все расходы указаны в евро.

Небольшое пояснение по бизнес-терминологии, хоть это и неважно: *Provident fund (фонд обеспечения) - это такая мера социальной защиты - социальное страхование и/или накопительный фонд для пенсии. Более точно, это инвестиционный фонд, который совместно учреждается работодателем и работником в качестве долгосрочных сбережений для поддержки работника после выхода на пенсию. Он также представляет собой пособия по социальному обеспечению на рабочем месте, предлагаемые работнику.* Но хватит об этом… Давайте уже к примеру.

пример 01c.go

```go
package main

import (
    "fmt"
)

type Worker interface {
    CalculateSalary() int
}

type Permanent struct {
    empId    int
    basicpay int
    pf       int // provident fund
}

type Contract struct {
    empId    int
    basicpay int
}

type Freelance struct {
    empId       int
    ratePerHour int
    totalHours  int
}

//salary of permanent employee is the sum of basic pay and pf
func (p Permanent) CalculateSalary() int {
    return p.basicpay + p.pf
}

//salary of contract employee is the basic pay alone
func (c Contract) CalculateSalary() int {
    return c.basicpay
}

//salary of freelancer
func (f Freelance) CalculateSalary() int {
    return f.ratePerHour * f.totalHours
}
```

```
/*
total expense is calculated by iterating through the Worker
slice and summing the salaries of the individual employees
*/
func totalExpense(s []Worker) int {
    expense := 0
    for _, v := range s {
        expense += v.CalculateSalary()
    }
    return expense
}

func main() {
    perm1 := Permanent{
        empId:    1001,
        basicpay: 2500,
        pf:       20,
    }
    perm2 := Permanent{
        empId:    1002,
        basicpay: 3000,
        pf:       30,
    }
    contr1 := Contract{
        empId:    2002,
        basicpay: 2400,
    }
    freelancer1 := Freelance{
        empId:       4001,
        ratePerHour: 30,
        totalHours:  120,
    }
    freelancer2 := Freelance{
        empId:       4003,
        ratePerHour: 45,
        totalHours:  80,
    }
    employees := []Worker{perm1, perm2, contr1, freelancer1, freelancer2}
    fmt.Printf("Total Expense Per Month €%d\n", totalExpense(employees))
}
```

Мы здесь объявили интерфейс Worker с единственным методом `CalculateSalary() int` .

В компании работают три типа сотрудников: штатные (они же постоянные) - Permanent, контрактники - Contract и фрилансеры - Freelance. Оплата штатных сотрудников включает зарплату и pf (что это такое объяснено выше, хотя это и неважно для нас), в оплату контрактников входит только зарплата, а оплата фрилансеров почасовая, причём у каждого из них своя ставка. Для каждого из типов сотрудников мы объявляем свой тип данных, каждый из которых реализует интерфейс Worker - в каждом из них объявлен метод `CalculateSalary() int` . Фактически можно говорить, что величина каждого из этих типов является величиной (интерфейсного) типа Worker, что наша программа с блеском и демонстрирует: мы создаём слайс []Worker, в который входят величины всех трёх типов, ой, нет, одного типа Worker, этот слайс получает функция `totalExpense(s []Worker) int` и чудно его обрабатывает. Ура.

Особо отмечу одну приятность. Вот заведём мы новый тип работника, назовём его Piecework (сдельщик - что сделал, то и получил), его зарплата состоит из оплаты сделанных работ, соответственно, объявляем тип

```
type Piecework struct {
    empId       int
    payments    []int
}
```

и определяем его метод

```
//salary of piecework
func (p Piecework) CalculateSalary() int {
    var sum int
    for _, v:= range p.payments {
        sum += v
    }
    return sum
}
```

После этого можем спокойно добавлять работников этого типа, не меняя в коде вообще ничего.

Так и сделаем - и получим
пример 01d.go

```go
package main

import (
    "fmt"
)

type Worker interface {
    CalculateSalary() int
}

type Permanent struct {
    empId    int
    basicpay int
    pf       int // provident fund
}

type Contract struct {
    empId    int
    basicpay int
}

type Freelancer struct {
    empId       int
    ratePerHour int
    totalHours  int
}

type Pieceworker struct {
    empId    int
    payments []int
}

//salary of permanent employee is the sum of basic pay and pf
func (p Permanent) CalculateSalary() int {
    return p.basicpay + p.pf
}

//salary of contract employee is the basic pay alone
func (c Contract) CalculateSalary() int {
    return c.basicpay
}
```

```
//salary of freelancer
func (f Freelancer) CalculateSalary() int {
    return f.ratePerHour * f.totalHours
}

//salary of piecework
func (p Pieceworker) CalculateSalary() int {
    var sum int
    for _, v := range p.payments {
        sum += v
    }
    return sum
}

/*
total expense is calculated by iterating through the Worker
slice and summing the salaries of the individual employees
*/
func totalExpense(s []Worker) int {
    expense := 0
    for _, v := range s {
        expense += v.CalculateSalary()
    }
    return expense
}

// function outputs the type and the value of any receiver of the Worker type
func describe(e Worker) {
    fmt.Printf("Type: %T \t value:%v\n", e, e)
}

func main() {
    perm1 := Permanent{
        empId:    1001,
        basicpay: 2500,
        pf:       20,
    }
    perm2 := Permanent{
        empId:    1002,
        basicpay: 3000,
```

```
        pf:       30,
    }
    contr1 := Contract{
        empId:    2002,
        basicpay: 2400,
    }
    freelanc1 := Freelancer{
        empId:       4001,
        ratePerHour: 30,
        totalHours:  120,
    }
    freelanc2 := Freelancer{
        empId:       4003,
        ratePerHour: 45,
        totalHours:  80,
    }
    piece1 := Pieceworker{
        empId:    5002,
        payments: []int{450, 250, 430, 700, 315},
    }
    employees := []Worker{perm1, perm2, contr1, freelanc1, freelanc2, piece1}
    fmt.Printf("Total Expense Per Month €%d\n\n", totalExpense(employees))

    fmt.Printf("Type of employees is %T\n", employees)
    fmt.Println("Employees:")
    for _, emp := range employees {
        describe(emp)
    }
}
```

Результат:

```
Total Expense Per Month €17295

Type of employees is []main.Worker
Employees:
Type: main.Permanent     value:{1001 2500 20}
Type: main.Permanent     value:{1002 3000 30}
Type: main.Contract      value:{2002 2400}
Type: main.Freelancer    value:{4001 30 120}
Type: main.Freelancer    value:{4003 45 80}
```

```
Type: main.Pieceworker   value:{5002 [450 250 430 700 315]}
```

И эти результаты дают нам пищу для очень интересных размышлений…

Интерфейсный тип - каков его смысл. Интерфейсный тип изнутри.

Ага, там в этом же коде добавлено кое-что ещё. Посмотрим чуть более подробно на наш список работников `employees`. Это переменная типа []Worker - слайс Worker'ов, причём Worker - это интерфесный тип. Ну, да, так и есть, именно это и выводит наша программа. А вот дальше начинается интересно… Оказывается, что каждый Worker имеет свой тип, и этот тип вовсе не Worker. Особенно явно показывает это функция describe, которая принимает величину типа Worker и выводит её тип и значение. И типы эти оказываются разными, да собственно и значения тоже различаются, и совсем не только значениями - там и структуры из двух int'ов, и структуры из трёх int'ов, и структура из int'а и слайса int'ов. В общем ожидаемо, но необычно, мы привыкли к другому. Но и какого-то особого дискомфорта не ощущается - всё выглядит довольно естественно, ведь все типы работников - и Permanent, и Contract, и Freelancer, и Piecework - это работники, Worker'ы, все они получают зарплату, и для всех них расходы на неё могут быть подсчитаны, пусть и разными способами для каждого вида работников. И если с обычными типами Go жесточайше следить за сооответствием типов величин (вспомним, как нам приходилось всё время приводить, например, int к int64 или float32 к float64 при использовании библиотечных функций, да и не только их), то с интерфейсными типами дело обстоит совсем иначе. Да оно и понятно, ведь интерфейсный тип (интерфейс, говоря грубее и короче) - это чисто список возможных действий, без их конкретной реализации, это одни глаголы, это объекты без свойств - без существительных и прилагательных. А вот конкретные типы, реализующие интерфейс Worker - Permanent, Contract, Freelancer, Pieceworker - имеют вполне конкретное содержание. Но вместе с этим они реализуют интерфейс Worker, фактически величины этих типов являются Worker'ами. Собственно, так и можно себе представлять - все Permanent'ы есть Worker'ы, также как и остальные типы работников. Интерфейсный тип поддерживает любое значение, которое реализует все его методы. Повторю, всё выглядит логично, никаких противоречий не видно, нет мозгового дискомфорта, всё хорошо. Так что всё правильно, хотя наши представления о жёстой типизации немного подвинулись. Это не сдвиг, это развитие :)

Примечание. Да, конечно, всё это называется *полиморфизм*, но, пожалуй, не стОит пока вводить лишние термины, их сейчас и без того много, а слова *полиморфизм* и *инкапсуляция*

(о которой тоже будет разговор - говоря о встроенных интерфейсах) вполне себе подождут до разговора об “ООР” на Go. Конец примечания

Ну, и естественно предположить, что каждая величина интерфейсного типа хранит в себе не только свои значения (конкретного типа), но и сам этот конкретный тип, реализующий данный интерфейс.

Как именно они хранятся - пока (да и вообще) не очень важно. Как до них добраться, кроме как выковырять с помощью fmt.Print (точнее, выковыривается с помощью fmt.Sprint, которая вернёт строку с названием конкретного типа) - поговорим чуть попозже, но не в этот раз.

А пока повторим, то, что у нас уже было - поиск в упорядоченной последовательности и сортировку,- но уже на фоне новых представлений. Построим сортировку интерфейсов и поиск в упорядоченной последовательности интефейсов. И это круто. Вообще, в ближайшее время будет довольно много контейнеров, бОльшая часть которых нам уже известна, но будет и кое-что новенькое :)

Но это ещё не всё…

Чего же нам ещё?! Мы и так уже получили возможность, которая, имея привычку к жёсткой типизации данных, кажется фантастичекой - мы можем собирать в однотипные структуры данные разных типов, ну, условно разных, в каком-то смысле всё-таки одинаковых, но собирать воедино (в массивы, в слайсы, в связные списки и т.д.) совсем уж разнотипные данные вообще нет смысла.

И это “ещё” может на первый взгляд показаться некоторым отступлением от появившихся возможностей - рассмотрим более гомогенные (однородные) коллекции, в которых мы собираем данные одного типа. Но! Одного интерфейсного типа. А это означает, что мы можем одной такой коллекцией окучивать самые разнообразные типы, которые, конечно, реализуют этот интерфейсный тип.

Тут по ходу появились пара важных терминов: коллекция и контейнер. Не сообо вдаваясь в обсуждение, всё-таки надо бы дать хотя бы некоторое понимание, что они означают, в каком смысле используются.

Коллекция в программировании —

```
программный объект, содержащий в себе, тем или иным образом, набор значений одного и
ли различных типов, и позволяющий обращаться к этим значениям.
    Коллекция позволяет записывать в себя значения и извлекать их. Назначение коллек
```

> **ции — служить хранилищем объектов и обеспечивать доступ к ним. Обычно коллекции испо льзуются для хранения групп однотипных объектов, подлежащих стереотипной обработке. Для обращения к конкретному элементу коллекции могут использоваться различные методы , в зависимости от её логической организации. Реализация может допускать выполнение отдельных операций над коллекциями в целом. Наличие операций над коллекциями во мног их случаях может существенно упростить программирование.**

Контéйнер в программировании —

> **тип, позволяющий инкапсулировать в себе объекты других типов. Контейнеры, в отличие от коллекций, реализуют конкретную структуру данных.**

Для понимания - в 5-м семестре у нас уже были некоторые структуры данных: разные списки, стек, очередь, бинарная куча, бинарное дерево поиска.

Обсудить эти понятия, видимо, стОит, но особо вдаваться здесь ни к чему, да и грань между коллекциями и контейнерами довольно тонкая, хотя и ясная, так что не будем вдаваться в эти тонкости.

А мы сейчас введём интерфейс `Ordered` (упорядоченный). Понятно, что он связан с упорядочениями, т.е. с сортировками и всем, что с этим связано, в частности, например, с бинарным поиском в упорядоченной последовательности.

```
type Ordered interface {
    // a.Before(b) returns true iff
    // a is located before b in an ordered sequence
    Before (b Ordered) bool
}
```

Подчеркну, `a.Before(b)` означает, что a обязательно идёт раньше b в упорядочении. Грубо, говоря, это означает что `a < b` , именно строго меньше, если бы мы упорядочивали элементы в порядке возрастания.

А теперь попробуем сделать что-то, связанное с сортировкой, переупорядочиванием целых чисел. Определим тип `Integer` , реализующий интерфейс `Ordered` :

```
type Integer int

func (a Integer) Before(b Integer) bool {
    return a < b
```

```
}
```

И тут-то мы наткнёмся на вполне обоснованное возражение от компилятора - форма функции

```
func (a Integer) Before(b Integer) bool
```

не соответствует описанию этой функции в объявлении интерфейса - в ней тип аргумента `Ordered`, а не `Integer`, да и, понятно, он и не может быть иным. Что делать? Понятно что - реализовывать метод `Before` в соответствии с его объявлением в интерфейсе:

```
type Integer int

func (a Integer) Before(b Ordered) bool {
    return a < b
}
```

Но теперь опять не срастается - невозможно сравнивать величину конкретного типа (типа `Integer`) с величиной интерфейсного типа `Ordered`. Особенно если учесть, что интерфейсный тип - это нечто абстрактное. Да, понятно, что во время выполнения программы функция `Before` будет получать параметр конкретного типа, реализующего интерфейс `Ordered`, но это же во время выполнения, а компилировать-то надо сейчас. И это сразу приводит нас к необходимости явно заявлять конкретный тип величины интерфейсного типа.

Type assertion. - Заявление типа.

Чтобы извлечь из величины интерфейсного типа содержащееся в ней значение (напомню, что, интерфейс, он же величина интерфейсного типа, содержит, кроме значения, также и конкретный тип этой величины, этого интерфейса) применяется **заявление типа** или, говоря по-простому **type assertion**.

Синтаксис заявления типа

выглядит так `i.(T)`, где `i` - это интерфейс, а `T` - конкретный тип.

И теперь наша проблема с реализацией метода (a Integer).Before() решается:

```
type Integer int
```

```
func (a Integer) Before(b Ordered) bool {
    return a < b.(Integer)
}
```

Конкретный тип величины `b` - Integer. И мы используем синтаксис `b.(Integer)`, чтобы извлечь её значение.

Заявление типа не есть приведение (преобразование) типа. Type assertion isn't type casting (conversion).

С преобразованиями типов мы сталкиваемся чуть ли не с первого-второго занятия (первого семестра) и, напомню, выглядит оно даже синтаксически иначе:

```
    i := 2222
    x := float64(i)
    n := uint(f)
```

или вот целая программа `02a.go`, в которой функция инициализации рандом-генератора rand.Seed требует аргумент типа int64.

```
package main

import (
    "fmt"
    "math/rand"
    "time"
)

func main() {
    rand.Seed(int64(time.Now().Nanosecond()))
    fmt.Printf("Int100: %d %d %d\n", rand.Intn(100), rand.Intn(100), rand.Intn(100))
}
```

И преобразование типов выполняет именно ПРЕОБРАЗОВАНИЕ, внутреннее представление данных при преобразовании типов изменяется.

Заявление типа НЕ ПРЕОБРАЗУЕТ внутреннее представление, при заявлении типа просто ИЗВЛЕКАЕТСЯ значение интерфейса. Отсюда и различие в синтаксисе.

А что будет, если мы криво исполним заявление типа?

Сейчас и посмотрим. Программа `02b.go`

```
import (
    "fmt"
)

type Ordered interface {
    Before (b Ordered) bool
}

type Integer int

func (a Integer) Before(b Ordered) bool {
    return a < b.(Integer)
}

func main() {
    d:= []Integer{3,1,0}
    var x Ordered
    for _, x = range d {
        fmt.Println(x.(float64))
    }

}
```

не компилируется, сообщая

```
./prog.go:24:16: impossible type assertion:
    float64 does not implement Ordered (missing Before method)
```

Распознать и предотвратить такую непрятность можно, используя совершенно естественным для Go синтакис:

```
    v, ok := i.(T)
```

Второе значение, возвращаемое заявлением типа - это и есть bool, сообщающий об успехе или неудаче при заявлении. Так что перепишем нашу программу так - программа ‘02c.go`:

```
package main
```

```
import (
    "fmt"
)

type Ordered interface {
    Before (b Ordered) bool
}

type Integer int

func (a Integer) Before(b Ordered) bool {
    return a < b.(Integer)
}

type Float float64

func (a Float) Before(b Ordered) bool {
    return a < b.(Float)
}

func main() {
    d:= []Ordered{Integer(310), Float(310.5)}
    for _, x:= range d {
        if v, ok:= x.(Integer); ok {
            fmt.Println(v)
        } else {
            fmt.Println("impossible type assertion")
        }
    }
}
```

выведет

```
310
0 impossible type assertion
```

В случае неудачного заявления типа возвращается (кроме false) нулевое значение.

И раз уж пошёл разговор о различении конкретного типа величины интерфейсного типа, то поставим сюда и switch по типам.

Type switch

Синтаксис переключателя типов, скажем так, совершенно аналогичен обычному оператору многовариантного выбора switch, только в качестве вариантов используются типы. И сравнивается конкретный тип интерфейса с предложенными вариантами типов. Собственно, это и есть все различия. Точный синтаксис чётко виден в примере `02d.go` :

```
package main

import (
    "fmt"
)

type Worker interface {
    CalculateSalary() int
}

type Permanent struct {
    empId    int
    basicpay int
    pf       int // provident fund
}

type Contract struct {
    empId    int
    basicpay int
}

type Freelancer struct {
    empId       int
    ratePerHour int
    totalHours  int
}

type Pieceworker struct {
    empId    int
    payments []int
}

//salary of permanent employee is the sum of basic pay and pf
func (p Permanent) CalculateSalary() int {
```

```
    return p.basicpay + p.pf
}

//salary of contract employee is the basic pay alone
func (c Contract) CalculateSalary() int {
    return c.basicpay
}

//salary of freelancer
func (f Freelancer) CalculateSalary() int {
    return f.ratePerHour * f.totalHours
}

//salary of piecework
func (p Pieceworker) CalculateSalary() int {
    var sum int
    for _, v := range p.payments {
        sum += v
    }
    return sum
}

func main() {
    perm1 := Permanent{
        empId:    1001,
        basicpay: 2500,
        pf:       20,
    }
    perm2 := Permanent{
        empId:    1002,
        basicpay: 3000,
        pf:       30,
    }
    contr1 := Contract{
        empId:    2002,
        basicpay: 2400,
    }
    freelanc1 := Freelancer{
        empId:       4001,
        ratePerHour: 30,
        totalHours:  120,
```

```
    }
    freelanc2 := Freelancer{
        empId:       4003,
        ratePerHour: 45,
        totalHours:  80,
    }
    piece1 := Pieceworker{
        empId:    5002,
        payments: []int{450, 250, 430, 700, 315},
    }
    employees := []Worker{perm1, perm2, contr1, freelanc1, freelanc2, piece1}
    for _, emp := range employees {
        switch emp.(type) {
        case Permanent:
            fmt.Printf("Постоянный работник. ID %d\n", emp.(Permanent).empId)
        case Contract:
            fmt.Printf("Контрактник. ID %d\n", emp.(Contract).empId)
        case Freelancer:
            fmt.Printf("Фрилансер. ID %d\n", emp.(Freelancer).empId)
        case Pieceworker:
            fmt.Printf("Работник со сдельной оплатой. ID %d\n", emp.(Pieceworker).em
pId)
        default:
            fmt.Println("Неопределённый тип")
        }
    }
}
```

выводит

```
Постоянный работник. ID 1001
Постоянный работник. ID 1002
Контрактник. ID 2002
Фрилансер. ID 4001
Фрилансер. ID 4003
Работник со сдельной оплатой. ID 5002
```

Выражение `switch emp.(type)` как раз и указывает на то, что это выбор по типам. И дальше идёт просто сравнение указанных типов с когкретным типом величины `emp` .

Да, что-то материала собралось изрядно, возможно надо будет думать о том, что какую-то

часть надо перенести на второе занятие, но без всего этого не складывтся хоть что-нибудь содержательное, а без него ничего не придумываеся на практику. Можно, вероятно, построить практку чисто на тестировании всяких свойств и способов работы с интерфейсами, но как-то это эммммм…

Так что всё-таки вернёмся к нашему интерфейсу Ordered и построим сортированную коллекцию. Реализовывать её будем на слайсе Ordered'ов. Вспомним, что `Ordered` - это интерфейс, вот такой:

```
type Ordered interface {
    Before(b Ordered) bool
}
```

Сортированная коллекция - это коллекция, которая постоянно поддеживает отсортированность, т.е. добавление любого элемента сразу ставит его на нужное место. Соответственно, сортированная коллекция обязательно должна иметь метод `Insert`, который вставляет элемент в коллекцию, ну, и для контроля нужен метод `Print` для вывода сортированной коллекции. Это, пожалуй и есть необходимый минимум. Иллюстрирует его программа `03.go`:

```
package main

import (
    "sorted"
)

type Integer int

func (a Integer) Before(b sorted.Ordered) bool {
    return a < b.(Integer)
}

func main() {
    data := []Integer{5, 8, 2, 4, 3, 2, 9, 7}
    var sortdata sorted.SortedCollection
    var x sorted.Ordered
    for _, x = range data {
        sortdata.Insert(x)
    }
    sortdata.Print()
```

```
}
```

Тип `Integer` - это просто `int`, в котором реализован интерфейс `Ordered`. Код предельно прозрачен и понятен.

А саму структуру `SortedCollection` мы реализуем в `package sorted`, что хорошо видно из кода. Реализуем мы эту структуру на слайсе `Ordered` ов, полный код приведён в файле `sorted.go`, вот его текст:

```
package sorted

import "fmt"

type Ordered interface {
    Before(b Ordered) bool
}

type SortedCollection []Ordered

func (c *SortedCollection) Insert(x Ordered) {
    *c = append(*c, 0)
    i := len(*c) - 1
    for i > 0 && x.Before((*c)[i-1]) {
        i--
    }
    // Insert x into i-th position of slice (*c)
    copy((*c)[i+1:], (*c)[i:])
    (*c)[i] = x
}

func (c SortedCollection) Print() {
    fmt.Println(c)
}
```

Методы реализованы максимально просто, вплоть до тупо - это умышленно, это возможность дать детям задания на практику.

Задания для самостоятельной работы.

Ну, во-первых, разобраться с интерфейсами, поиграться с ними, с заявлением типа (type

assertion), со type switch. И, кроме того, несколько более точных и конкретных заданий, связанных с `SortedCollection`.

1. Изменить метод `(*SortedCollection) Insert(x Ordered)` так, чтобы исполнялся бинарный поиск места, куда вставлять x. Сравнить новый метод со старым по скорости с помошью benchmark.
2. И сразу в тему: написать ещё два метода
   `(SortedCollection) First(x Ordered) (index int, detected bool`
   и
   `(SortedCollection) Last(x Ordered) (index int, detected bool`
   которые находят место, соответственно, первого и последнего элемента эквивалентного x в сортированнной коллекции. detected - это индикатор, найден такой элемент или нет.
   Поясним смысл термина “эквивалентного”. Грубо говоря, это означает, равного x. Но всё-таки эквивалентность - это что-то другое. Ведь два элемента, которые могут стоять в упорядочении в произвольном порядке не обязательно должны быть полностью равны. Например, мы сортируем список людей по дате рождения - имена, фамилии и прочее у людей, которые родились в один день, могут отличаться, но с точки зрения упорядочения они одинаковы. Т.е две величины
   x и y эквивалентны, если выполняется условие `!x.Before(y) && !y.Before(x)`.
3. Удалить элемент x из сортированнной коллекции. Тут стОит уточнить: во-первых, речь идёт об элементе, эквивалентном x; во-вторых, хорошо бы уточнить, удалять первый такой элемент, удалять последний такой элемент, удалять все такие элементы или имеется в виду что-то ещё. Возможно есть смысл написать все эти варианты. Разумеется, это надо оформить в виде метода (методов).
4. Определить ещё какой-нибудь тип, реализующий интерфейс `Ordered` и отсортировать массив элементов этого типа, собрав из них сортированную коллекцию. Ну, например, тот же пример с людьми, которых мы сортируем по дате рождения. Или, например, структуры, в которых указаны спортсмены и количество набранных ими очков. Сортировать, конечно, надо по убыванию количества очков - нужна соответствующая реализация метода `Before`.